Engineering Science and Technology, an International Journal xxx (2016) xxx–xxx

HOSTED BY

ELSEVIER

Contents lists available at ScienceDirect

Engineering Science and Technology, an International Journal

journal homepage: www.elsevier.com/locate/jestch

Full Length Article

A novel approach for deriving interactions for combinatorial testing

Sangeeta Sabharwal, Manuj Aggarwal *

Department of Computer Engineering, NSIT, New Delhi, India

ARTICLE INFO

Article history:
Received 15 December 2015
Revised 20 April 2016
Accepted 1 May 2016
Available online xxxx

Keywords:
t-way testing
Combinatorial testing
Interaction faults
Data flow techniques
DD path graph
Interaction testing

ABSTRACT

Combinatorial testing focuses on identifying faults that arise due to interaction of values of a small number of input parameters. Also known as t-way testing, it reduces the size of test set by selecting a minimal set of test cases that cover all the possible t-way tuples. An optimal value of t (degree of interaction) for t-way testing for the system would maximize fault detection count in minimal number of test cases. However, identification of an optimal value of for t-way testing for the system remains an open issue. In this paper, we present an approach to identify the interactions that exist in the source code, thereby reducing the count of interactions to be tested. DD path graph is generated from the source code and interactions are identified using data flow techniques. Two case studies are also discussed in order to demonstrate our approach. Experimental results indicate that our approach significantly reduces the count of interactions to be tested without significant loss of fault detection capability. The approach is extensible to large sized structured programs.

1. Introduction

Software testing is an expensive and time consuming activity that leads to production of reliable software systems [1,2]. Due to its importance, testing process is allocated a large share of the software development resources [3]. However, it is often observed that when the usage of large data-intensive software increases, the modules which have passed conventional testing methods start developing undetected errors [4]. The possible reasons include addition of records with an oddball combination of values that has not occurred before in the software. It is observed that these rare combinations of values which have escaped testing process and usage of software can cause interaction failures. To avoid such failures, it is desirable to test all combinations of values in an exhaustive manner. However, exhaustive testing is not feasible either due to time or resources availability. Thus a technique is required that focuses on testing combination of values.

Combinatorial (t-way) testing focuses on testing combinations of values. It is based on the observation that a large number of faults are caused by interactions of a few input parameters. Hence rather than testing all combinations in an exhaustive manner, combinations of only few parameters are tested. In order to generate test set, values for input parameters are selected such that every possible combination of values of any t parameters occurs at least once [5]. t is also known as the strength of coverage or interaction strength.

As an example, let us consider 3 input parameters, A, B and C, each can have 2 possible values, 0 and 1. Pairwise testing (where $t = 2$) would require the following 4 test cases as given in Table 1. Test cases are designed such that all possible pairs of values are getting covered.

As per studies, it is observed that maximum value of interaction strength is 4–6 for most of the systems [4]. As the value of interaction strength increases, the total number of detectable errors increases. But, an increase in interaction strength leads to an increase in the test set size, and hence increases the cost of testing. On the other hand, lower interaction strength leads to reduction in test set size which affects faults detection rate. Thus an optimal value of interaction strength can substantially reduce the testing costs without compromising fault detection capability. However, not much research is done in this area.

We have focused on two types of interaction failures as defined in the literature [2]. These are type 1 interaction failures and type 2 interaction failures. Type 1 interaction failures occur when a code segment in which a fault exists is executed. Due to interaction among variables, the faulty code is executed. For the pseudo code given in Fig. 1, a software system is observed to fail only for a set of customers residing at a particular location. Due to an interaction of two or more variables, a block of code is executed in which fault exists. Type 2 interaction failures occur when performing some computation on two or more variables leads to an incorrect result.

* Corresponding author.
E-mail addresses: ssab63@gmail.com (S. Sabharwal), mmanuj.aggarwal@gmail.com (M. Aggarwal).

Peer review under responsibility of Karabuk University.

http://dx.doi.org/10.1016/j.jestch.2016.05.008
2215-0986/

Table 1
Pairwise test set for a problem where each variable can have 2 possible values.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

```
Begin
if (customer belongs to set A){
    //some code here
    if(customer.residence==UK){
        //faulty code here
    }
    else{
        //block of code-executes normal
    }
}
End
```

Fig. 1. Pseudo code illustrating type 1 interaction failure.

Type 2 interaction failures are illustrated using pseudo code in Fig. 2. Here, placing an erroneous operator causes the set of variables involved in computation to produce an incorrect result. In this paper, we aim to identify interactions that may cause these two types of interaction failures to occur.

Data flow analysis techniques derive the information about the flow of data that exist in the program [6]. At each step in the program, the information about definition and usage of variables is obtained. The two usages defined are computation use (c-use) and predicate use (p-use) [2]. The usage of a variable is considered c-used if the variable occurs as a part of an assignment statement, as an argument passed to a function, in a subscript expression or in an output statement. The usage of a variable is considered p-used if the variable is used in a condition expression.

In this paper, a novel approach is presented that derives the interactions of variables that exist in the code. We have extended an existing work [7] by modifying their approach to handle the modularized programs where the basic modules are functions. The previous approach to identify the interactions among variables was meant for programs consisting of a single function only. Further, we have optimized the approach so that the resulting flow graph created using our approach contains less number of nodes as compared to the previous work. A flow graph is generated from the source code and data flow technique is applied on it. The c-use of a variable is redefined for the approach. From the flow graph and usage of variables, interactions are identified. The approach achieves a significant reduction in the count of interactions to be tested. As a result, rather than testing all the possible t-way interactions, only the identified interactions are tested which leads to reduction in testing costs. From the literature, the authors could not find any study that tries to identify the strength of coverage for a large sized system. Hence, our study may be considered as unique.

```
Begin
float x, y, z;
float result=(x*y)/z// it should be (x*y)-z
//block of code
End
```

Fig. 2. Pseudo code illustrating type 2 interaction failure.

The remainder of this paper is organized as follows. Next Section discusses the related work done in this field. In Section 3, the proposed approach to identify the interactions is explained. In Section 4, we have illustrated our approach with the help of case studies. Section 5 discusses the experimental results. In Section 6, threats to validity are given. Finally, in Section 7, some conclusions and future work are outlined.

2. Related work

Combinatorial testing (CT) has been widely studied and has become a well-accepted testing method. A large number of research articles have focused on CT. Researchers [8] have broadly classified them into eight categories. Category 1 includes all the articles that focus on generation of combinatorial test suites. Most widely Covering Arrays [9] are generated using metaheuristic methods such as Tabu Search, Simulated Annealing [10], Ant Colony Optimization, Particle Swarm Optimization, simplified swarm optimization [11] and Genetic Algorithms [12]. However, some authors have generated test suite using greedy methods [5], mathematical methods and recursive methods. A few researchers have taken into account the seeding and constraints. Category 2 includes all the articles that focus on test case prioritization. In CT, prioritization has been achieved either by reordering an existing test suite on the basis of prioritization criteria (coverage measurement, etc.) or generating prioritized test suite that includes important combinations first. Researchers [13] have designed formulas to compare the weights of the prioritization. Category 3 and 4 includes articles that evaluate how CT has improved the software quality and articles that focus on metrics. Most of the articles have considered combinatorial coverage as metric to evaluate the effectiveness of combinatorial testing. Category 5 includes the articles that study the application of CT to various types of applications. CT has been used for performance evaluation [14], feature testing of mobile phone applications [15,16], testing of network interface [17], etc. Category 6 includes articles that have taken into account the constraints [18]. A test suite must satisfy constraints and invalid test cases need to be eliminated.

In category 7, articles focusing on fault detection are discussed. Here, techniques to identify faults are discussed that have caused a failure to occur. It is required to identify the faults and remove them so as to improve the software quality [19]. Classification tree approach [20], classification and tuple relationships [21] are used for identification of failure inducing combinations. However, these approaches can identify the faults only when failures have occurred.

Last category includes all the articles that focus on modeling of System Under Test (SUT). Although building a precise model of SUT, such as Input Parameter Model [22] leads to effective CT, not much research has been done in this area. An SUT model for CT includes parameters and their values. It also includes the relationships among the parameters. These elements are identified from various documents generated during software life cycle, such as SRS, SDD, implementation, etc. Some researchers have identified parameters and their values from artifacts such as UML activity diagram [23] and UML sequence diagram [24]. Attempts have been made to reduce the size of test suite by reducing the count of parameters for SUT model [25] and identifying relationships between input and output parameters [26].

As can be concluded from the literature, CT has been extensively studied. Various research articles exist that focus on CT. As stated earlier, there exists a need to identify all the possible interactions among variables for t-way testing. However, not much research has been done to identify the strength of coverage for a large sized system. Some approaches, as mentioned in category 7

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

of related work, attempt to identify the failure inducing combinations. However, these approaches can identify the faults only when failures have occurred. Thus, there exists a need for an approach that identifies all the possible interactions among variables that exist in a large sized system. The approach must be able to identify faults even when the failure has not occurred.

3. Proposed approach

In this section the proposed approach that identifies interactions of variables from a structured program is discussed. By structured program we mean a program that follows structured paradigm, i.e. the program is divided into functions. The approach uses data flow concepts to compute the usage of variables to identify interactions among variables. The input of the approach is source code and the output generated is a set of interactions that exist in the source code. A Decision to Decision (DD) path graph is used, where each node corresponds to a sequence of statements and each edge corresponds to the directed flow of control among the nodes in the program [2]. The in-degree of a node is defined as the count of edges entering a node. Similarly, the out-degree of a node is defined as the count of edges leaving a node. As the focus of our approach is to identify interactions among variables, c-use is redefined as follows. For the proposed approach, a node of a DD path graph is considered a c-use of a variable, if and only if, the variable occurs as a part of an assignment statement or in a subscript expression. In order to handle functions, a new usage, named as return-type use (r-use) is defined. A node of a DD path graph is r-use node for a function *f*, if and only if, there exists a function call to *f* at the statement corresponding to that node. Further, if the value returned by the function being called, represented as rv(*f*) is c-used, then the statement is considered a c-use for the return value of the function. Similarly, if the value returned by the function being called rv(*f*) is p-used, then the statement is considered a p-use for the return value of the function. However, if the function returns void or the value returned by the function is not used, then the function is not considered for c-use or p-use. In this approach, we have considered rv(*f*) as a separate variable and identified its interactions with other variables that exist in the source code.

In the proposed approach, the interactions identified from the source code may be of different strengths. Based on the existing work on identification of interactions among variables [7], the main steps are discussed below.

3.1. Transformation of source code into DD path graph

The source code of the program is transformed into DD path graph [27]. A node is created for a block of statements rather than a node for each statement. A block consists of sequence of statements having sequential flow of control. As the program is organized into hierarchy of functions, for each function a DD path graph is created. A DD path graph is also created for the main module. Next, the usage of variables and functions at block level is computed. As per [27], high quality commercial tools are available which generate DD flow graph of a given program. The time complexity of this step is in O(n), where n is total number of statements.

3.2. Computation of usage at statement level

For each statement, the c-use and p-use for the variables and r-use for the functions are computed. P-use of a variable occurs in conditional statement or loop statement. If a variable is p-used, we identify the statement number at which corresponding condition block or loop block ends and name it as scope number of the variable for that statement number. In case of r-use, we further classify the usage of the value returned by the function f, denoted as rv(f), to c-use or p-use. However, if the function returns void or the value returned by the function is not used, then the function is not considered for c-use or p-use. It is required to traverse the complete program in order to identify variables and identify their usage. The time complexity of this step is in O(n), where n is total number of statements.

At the completion of this step, we obtain 3 sets for each statement: set 1 contains variables that are c-used at that statement; set 2 contains variables that are p-used at that statement and a set 3 contains functions that are r-used at that statement.

3.3. Computation of usage at block level

A node corresponds to a block of sequential statements. To identify the usage at a node, a node is associated with the usage data identified for the statements that correspond to that node.

At the completion of this step, we obtain 3 sets of sets: a set of sets of variables that are c-used at that node, a set of sets of variables that are p-used at that node and a set of sets of functions that are r-used at that node. The time complexity of the above step is in $O(n_{DD})$, where n_{DD} is number of nodes in DD path graph and n is total number of statements ($n_{DD} < n$).

For instance at statement 1, variables ‘a’ and ‘b’ are c-used, at statement 2, variable ‘b’ is c-used and at statement 3, variables ‘b’ and ‘c’ are c-used. If these three statements belong to a block b1, then c-uses for the block are {{a, b}, {b}, {b, c}}.

3.4. Reduction rules for DD path graph

In order to make our approach handle large sized programs, we aim to reduce the count of nodes and edges that exist in DD path graph. Nodes where no usage (c-use, p-use or r-use) exists can be eliminated from the graph. A node is eliminated such that the overall structure of the graph is not modified. Elimination is done only if the count of independent paths does not change. Certain rules have been defined that decide whether a node can be deleted from the graph [7]. These rules are given below:

4.1. If the in-degree and the out-degree of a node are 1, then the node can be removed by creating an edge by joining its predecessor node to its successor node.
4.2. If the in-degree of a node is 0 and the out-degree of the node is 1, then the node is start node and can be removed from the graph.
4.3. If the in-degree of a node is 1 and the out-degree of the node is 0, then the node is end node and can be removed from the graph.
4.4. If the in-degree of a node is greater than 1 and out-degree of the node is 1, then the node is a merge node and can be removed by connecting all the incoming edges of the node to its successor node.
4.5. If the in-degree of a node is 1 and out-degree of the node is greater than 1, then the node is a branch node and can be removed by connecting all the outgoing edges of the node to its predecessor node.

After all the possible elimination of nodes, parallel edges of same direction that exist between two nodes are replaced by a single directed edge of the same direction between those nodes of the graph. The resulting graph obtained is known as Reduced DD path Graph (RDD path graph). The above procedure is represented in algorithm 1. Let the number of nodes in RDD path graph be n_{RDD} ($n_{RDD} < n_{DD}$) and number of edges be e_{RDD}. Lines 5–10 of this

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

algorithm require $O(n_{RDD})$ steps and lines 11–14 of the algorithm requires $O(e^2_{RDD})$ steps. Removal of node at line 7 can be performed in O(1) time if pointer to its parent is stored. Similarly, removal of edge from edge list at line 14 can be performed in O(1) time. The execution count of while loop at line 2 does not depend on number of statements in the program. For our case studies, it is executed two times and generally executed at most 3–4 times. Thus, the time complexity of the algorithm is in $O(n_{RDD} + e^2_{RDD})$. Rather than working on a DD path graph having large number of nodes, we work on RDD path graph having less number of nodes.

Algorithm 1. Algorithm for creating RDD path graph from DD path graph using rules
Input: DD path graph.
Output: RDD path graph.

```
begin
1. boolean flag=true;
2. while(flag==true)
3. {
4.    flag=false;
5.    for each node in the graph { //nodes can be in any order
6.       if no usage exists {
7.          remove the node if any of the rules (4.1-4.5) satisfies;
8.          flag=true;
9.       }
10.   }
11.   for each edge e_i
12.      for each edge e_j having e_i.source=e_j.source
13.         if (i≠j and e_i.destination=e_j.destination)
14.            remove edge e_j from the edge's list
15. }
end
```

3.5. Identification of interactions

After performing the reductions on the DD path graph, the next step is to identify the interactions. The resulting RDD path graph contains the merge nodes, branch nodes and nodes at which at least one of the variables is c-used or p-used or at least one of the functions is r-used. For identifying different types of interactions, the procedures are discussed below.

3.5.1. Identification of interactions that can cause type 1 interaction failures

Here we aim to identify the set of variables that are p-used in a path. The same procedure is to be repeated for each possible path that exists in the graph. Each set of variables represents an interaction. In order to achieve this, we identify independent paths for the graph. Then each independent path is traversed once. An empty set is defined for each path to store set of set of variables. We have defined scope number for a decision node as node number at which corresponding condition block or loop block ends. It can be used to identify nested loops. While traversing the nodes of the path, all the variables that are p-used at any node in the path are to be added to the set. A variable that is p-used at a node is added to the current set if current node number is lesser than the scope number of the previously added variable; else it is added to the new set. The above procedure is repeated for each RDD path graph obtained from the source code. Total number of independent paths in a graph is: e_{RDD}-n_{RDD} + 2. For loop at line 7 is executed e_{RDD}-n_{RDD} + 2 times and for loop at line 9 is executed n_{ip} times, where n_{ip} is number of nodes in independent path. Comparison at line 13 will take constant time of O(1). Also, addition to the subset can be performed in O(1) time. Thus, total steps are $O(n_{RDD}{}^*n_{ip})$. As $O(n_{RDD}{}^*n_{ip}) < O(n^2_{RDD})$, the time complexity of the above step is in $O(n^2_{RDD})$.

Algorithm 2. Algorithm for identification of interactions that may cause type 1 interaction failures
Input: RDD path graph, and set of set of variables that are p-used.
Output: set of type 1 interactions.

```
begin
1. Initialize IP = number of independent paths that exist in the
   graph;
2. let S = {S_1, S_2, S_3, ....., S_IP}, where S_i (1 ⩽ i ⩽ IP) represents
   set of sets of variables that are p-used for independent path
   i;
3. for each set of set of variables i, in S set
4. begin
5.    S_i={ } be an empty set of set;
6. end
7. for each independent path, i
8. begin
9.    for each node of the path
10.   begin
11.      if (p-use of the node is not empty)
12.      then
13.         if (scope number of previously added
   variable > current node number)
14.            add the variables to the current subset of the set
   S_i;
15.         else
16.            add the variables to the new subset of the set S_i;
17.      end;
18.   end;
19. end;
20. end;
```

The above procedure identifies a set of set of variables for each path. Each set obtained is a probable interaction that can cause type 1 interaction failure of that code. The above procedure is represented in algorithm 2. We did not explicitly handled loops. However, as we have taken independent paths, a loop is traversed at least once.

3.5.2. Identification of interactions that may cause type 2 interaction failures

Here we aim to identify a set of variables that are c-used at a statement. The same set is to be identified for all the statements of the program. The desired set of variables is already calculated in step 2 of the proposed approach. Each set of variables represents an interaction existing at any statement of the program that may cause type 2 interaction failures.

3.6. Minimising the set of interactions

By this step, we have obtained two sets of interactions: a set of probable interactions to cause type 1 interaction failures and a set of probable interactions to cause type 2 interaction failures. The union of these two sets results into a set of interactions that exist in the source code. However, the resultant set may have redundant interactions. As illustration, assuming a system is taking 7 input variables (a, b, c, d, e, f and g). On applying our approach, assume we obtain the following interactions: {(a, c), (a, d), (d, g), (f, g), (d, e) and (a, c, d)}. It can be noticed that interaction (a, c, d) is a 3-way interaction and 2-way interactions {(a, c), (a, d), (c, d)} are implicitly covered by it. Hence, the interactions (a, c) and (a, d)

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

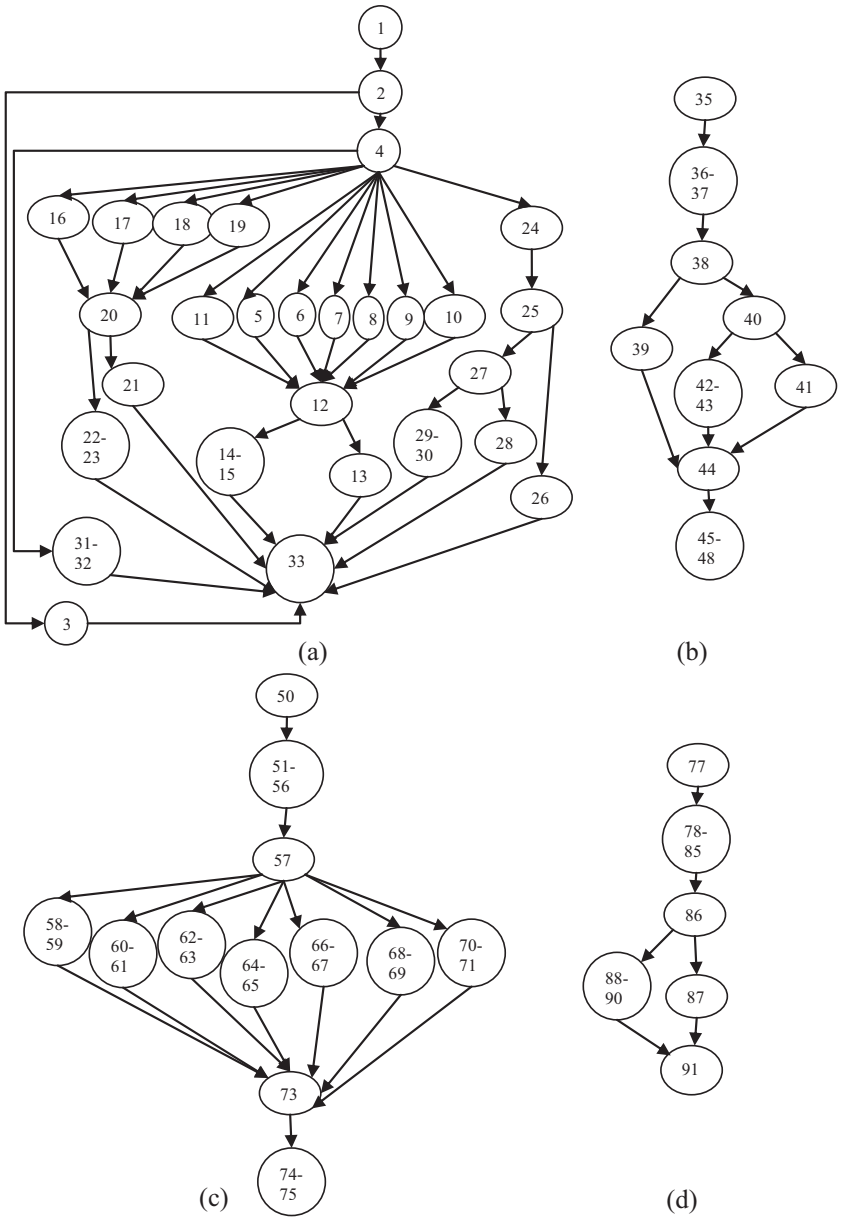

Fig. 3. DD path graph created from the function (a) int isdatevalid(int month, int date, int year) (b) int fm(int date, int month, int year) (c) int day_of_week(int date, int month, int year) and (d) void main().

are eliminated from the minimal set. As a result, the minimal set of interactions that exists in the source code is identified using the proposed approach. For each interaction, this step requires searching the list of interactions. Deletion from the list can be performed in O(1) time. Hence, time complexity is in $O(n_{int}^2)$, where n_{int} is total number of interactions.

It is to be noted that in our proposed approach, rather than combining DD path graphs of the functions with the main program, we have considered each DD path graph independently. Moreover, we have created a node for a block of statements rather than a node for each statement. Thus our approach is extensible to large sized programs. Further, rather than considering functions arguments as a means of communication between the function and the main program, we have considered the value returned by a function as a means of communication by the function to the main program. Lastly, as we are considering a single statement at a time, increasing the number of parameters would not affect the approach. This makes our approach scalable for programs involving large number of parameters.

The point of difference of our approach from the data flow analysis is that in our approach, c-use and p-use are computed at statement level rather than at block level. Moreover, at block level, two sets of set of variables are obtained, one for the set of variables that are c-used and other for the variables that are p-used. Whereas in case of data flow analysis, two sets of variables are obtained, one for the variables those are c-used and other for the variables that are p-used.

4. Case study

In order to demonstrate the effectiveness of the proposed approach, two case studies have been used. These problems have been widely adopted in other researchers' work [6]. Case study 1 is a C program that accepts Year, Month and Date from the user and displays the day of the month. Case study 2 is C program that accepts various employee details and calculates income tax. Next, we will discuss each case study in detail and identify interactions among variables using our proposed approach.

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

Table 2
c-use, p-use and r-use of variables at each statement of the case study 1.

Statement	c-use	p-use	r-use
2		Date	
4		Month	
12		Date	
20		Date	
25		Date	
27		Date	
38		Year	
40		Year	
44	Leap, month		
46	Fmonth		
53	rv(fm(date, month, year))		fm(date, month, year)
54	Year, date, ret_fm		
56	Dow		
57		Dow	
86		rv(isdatevalid(date, month, year))	isdatevalid(date, month, year)
89	rv(day_of_week (date, month, year))		day_of_week (date, month, year)

Table 3
c-use, p-use and r-use of variables at each node of the DD flow graph of case study 1.

Node	c-use	p-use	r-use
2		Date	
4		Month	
12		Date	
20		Date	
25		Date	
27		Date	
38		Year	
40		Year	
44	Leap, month		
45–48	Fmonth		
51–56	{{rv(fm(date, month, year))}, {year, date, ret_fm}, {dow}}		fm(date, month, year)
57		Dow	
86		rv(isdatevalid (date, month, year))	isdatevalid (date, month, year)
88–89	rv(day_of_week(date, month, year))		day_of_week (date, month, year)

4.1. Case study 1: program to display the day of the month

Appendix contains the source code of the case study 1 [28] that accepts Year, Month and Date from the user and displays the day of the month. The program is taken as input for the proposed approach. The program has 7 input parameters: namely date, month, year, fmonth, leap, dow and ret_fm. Four functions defined in the program are int isdatevalid(int month, int date, int year), int fm(int date, int month, int year), int day_of_week(int date, int month, int year) and void main().

The source code is transformed into the DD path graph. Fig. 3a represents the DD path graph for the function int isdatevalid (int month, int date, int year), Fig. 3b for the function int fm(int date, int month, int year), Fig. 3c for the function int day_of_week (int date, int month, int year) and Fig. 3d for the function void main (). At each statement, the c-use and p-use of the variables and r-use of the functions are identified and are shown in Table 2.

Next, the usage of variables and functions at block level is computed. The statements 53, 54 and 56 are part of the block (51–56). The set of variables that are c-used at statement 53 is {rv(fm(date, month, year))}, at statement 54 is {year, date, ret_fm} and at statement 56 is {dow}. When computing the usage at block level, the set of sets of variables that are c-used at block (51–56) is {{rv(fm(date, month, year))}, {year, date, ret_fm}, {dow}}. Table 3 shows the c-use and p-use of the variables and r-use of the functions identified at each node.

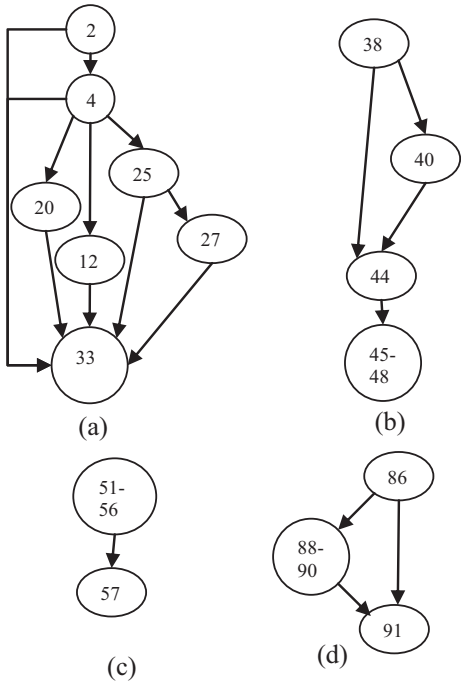

Fig. 4. RDD path graph created from the DD path graph of the function (a) int isdatevalid (int month, int date, int year) (b) int fm(int date, int month, int year) (c) int day_of_week (int date, int month, int year) and (d) void main().

Further, using reduction rules as given in Section 3, the graph is transformed to an RDD path graph. Let us consider DD path graph given in Fig. 3(a). Let us consider node 1. Here no usage exists. Hence using rule 4.2 node 1 can be removed. Next, for node 2, as usage (p-use) exists, it can't be eliminated and no rule is checked. Similar is the case with node 4. Then at node 16, no usage exists. Using rule 4.1, node 16 is removed and its predecessor node 4 is connected to its successor node 20. Similarly, in Fig. 3(c), using the algorithm 1 for reduction, in first iteration, nodes 58–59, 60–61, 62–63, 64–65, 66–67, 68–69 and 70–71 are removed using rule 4.1. In this iteration, either node 73 is removed using rule 4.4 or node 74–75 is removed using rule 4.3. Further all the parallel edges are removed. In second iteration, either node 73 or 74–75 is removed using rule 4.3. Fig. 4a represents the RDD path graph for the function int isdatevalid (int month, int date, int year), Fig. 4b for the function int fm(int date, int month, int year),

Table 4
Interactions among variables that may cause type 1 interaction failures in case study 1.

Path	Interactions
2, 4, 12, 33	Date, month
2, 4, 20, 33	Date, month
2, 4, 25, 27, 33	Date, month
2, 33	Date
2, 4, 25, 33	Date, month
2, 4, 33	Date, month
38, 40, 44, 45–48	Year
38, 44, 45–48	Year
51–56, 57	Dow
86, 88–90, 91	isdatevalid(date, month, year)
86, 91	isdatevalid(date, month, year)

Table 5
Interactions among variables that may cause type 2 interaction failures in case study 1.

Interactions
Leap, month
Fmonth
Date, year, ret_fm
Dow
day_of_week(date, month, year)
fm(date, month, year)

Table 6
Minimal set of interactions among variables in case study 1.

Interactions
Date, month
Leap, month
Date, year, ret_fm

Fig. 4c for the function int day_of_week (int date, int month, int year) and Fig. 4d for the function void main().

From the RDD path graph, independent paths are computed and probable interactions to cause type 1 interaction failures are identified. For each path, interactions are identified using algorithm 2 and are shown in Table 4. To identify interactions that can cause type 2 interaction failures, c-use of the variables at each node of the graph is considered.

The c-uses of the variables have already been computed at each node of the graph and are shown in Table 3. The set of interactions that may cause type 1 and type 2 interaction failures is shown in Tables 4 and 5 respectively. Finally, the set of interactions identified is reduced to a minimal set, which is shown in Table 6.

In order to verify that reduction of graph does not lead to loss of interactions, let us consider DD path graph given in Fig. 3(a) and its reduced RDD path graph given in Fig. 4(a). Table 7 shows the generated independent paths for both graphs and type 1 interactions calculated using these paths. As can be seen, DD path graph and RDD path graph generate same interactions. The reduction helps to achieve smaller length paths.

4.2. Case study 2: program to calculate income tax

We have developed the source code (provided in Appendix A) of income tax problem (for demonstration purpose only) that accepts employee's name, ID, Gender, Basic Salary (Bsal), DA (Dsal), HRA (Hsal), age, experience in years (exp_in_yrs), extra_working_hours, department (dept), designation (design), pf and insurance amount and displays the tax calculated. The program is taken as input for the proposed approach. The program has 19 parameters: namely employee's name, ID, Gender, Bsal, Dsal, Hsal, age, exp_in_yrs, extra_working_hours, dept, design, pf, insurance amount, total salary, total income, tax, vc, rv_calc_vc and rv_calctax. Five functions defined in the program are float calc_vc(int age, int exp_in_yrs, int extra_working_hrs, char dept[]), void insert(), void display(), float calctax(float totalincome) and int main().

At each statement, the c-use and p-use of the variables and r-use of the functions are identified and are shown in Table 8.

The source code is transformed into the DD path graph. Fig. 5a represents the DD path graph for the function float calc_vc(int age, int exp_in_yrs, int extra_working_hrs, char dept[]), Fig. 5b for the function void insert(), Fig. 5c for the function void display(), Fig. 5d for the function float calctax(float totalincome) and Fig. 5e for the function int main().

Next, the usage of variables and functions at block level is computed and are given in Table 9. Further, using reduction rules as given in Section 3, the graph is transformed to an RDD path graph. Fig. 6a represents the RDD path graph for the function float calc_vc (int age, int exp_in_yrs, int extra_working_hrs, char dept[]), Fig. 6b for the function void insert(), Fig. 6c for the function void display(), Fig. 6d for the function float calctax(float totalincome) and Fig. 6e for the function int main(). From the RDD path graph, independent paths are computed and probable interactions to cause type 1 interaction failures are identified.

For each path, interactions are identified using algorithm 2 and are shown in Table 10. To identify interactions that can cause type 2 interaction failures, c-use of the variables at each node of the graph is considered.

The c-uses of the variables have already been computed at each node of the graph and are shown in Table 8. The set of interactions

Table 7
Interactions among variables for the function int isdatevalid (int month, int date, int year) that may cause type 1 interaction failures.

Independent graphs for Fig. 3(a)	Independent graphs for Fig. 4(a)	Interactions
1, 2, 4, 24, 25, 27, 29–30, 33	2, 4, 25, 27, 33	Date, month
1, 2, 4, 24, 25, 26, 33	2, 4, 25, 33	Date, month
1, 2, 4, 11, 12, 14–15, 33	2, 4, 12, 33	Date, month
1, 2, 4, 5, 12, 13, 33		
1, 2, 4, 5, 12, 14–15, 33		
1, 2, 4, 6, 12, 14–15, 33		
1, 2, 4, 7, 12, 14–15, 33		
1, 2, 4, 8, 12, 14–15, 33		
1, 2, 4, 9, 12, 14–15, 33		
1, 2, 4, 10, 12, 14–15, 33		
1, 2, 4, 31–32, 33	2, 4, 33	Date, month
1, 2, 3, 33	2, 33	Date
1, 2, 4, 16, 20, 22–23, 33	2, 4, 20, 33	Date, month
1, 2, 4, 17, 20, 21, 33		
1, 2, 4, 18, 20, 21, 33		
1, 2, 4, 19, 20, 21, 33		

Table 8
c-use, p-use and r-use of variables at each statement of case study 2.

Statement	c-use	p-use	r-use
7		Dept	
8	Bsal		
10	Bsal		
11	Vc, bsal, extra_working_hours		
12		Age	
13		Exp_in_yrs	
14	Vc, dsal		
15		Design	
17	Vc		
43			rv_calc_vc()
44	Bsal, dsal, hsal, vc		
45	Totalsalary,pf,insurance		
55		Gender	
56		Totalincome	
60		Totalincome	
62	Totalincome		
64		Totalincome	
66	Totalincome		
68		Totalincome	
70	Totalincome		
74	Totalincome		
77		Gender	
79		Totalincome	
83		Totalincome	
85	Totalincome		
87		Totalincome	
89	Totalincome		
91		Totalincome	
93	Totalincome		
97	Totalincome		
104			rv_insert()
105			rv_display()
106			rv_calc_tax()

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

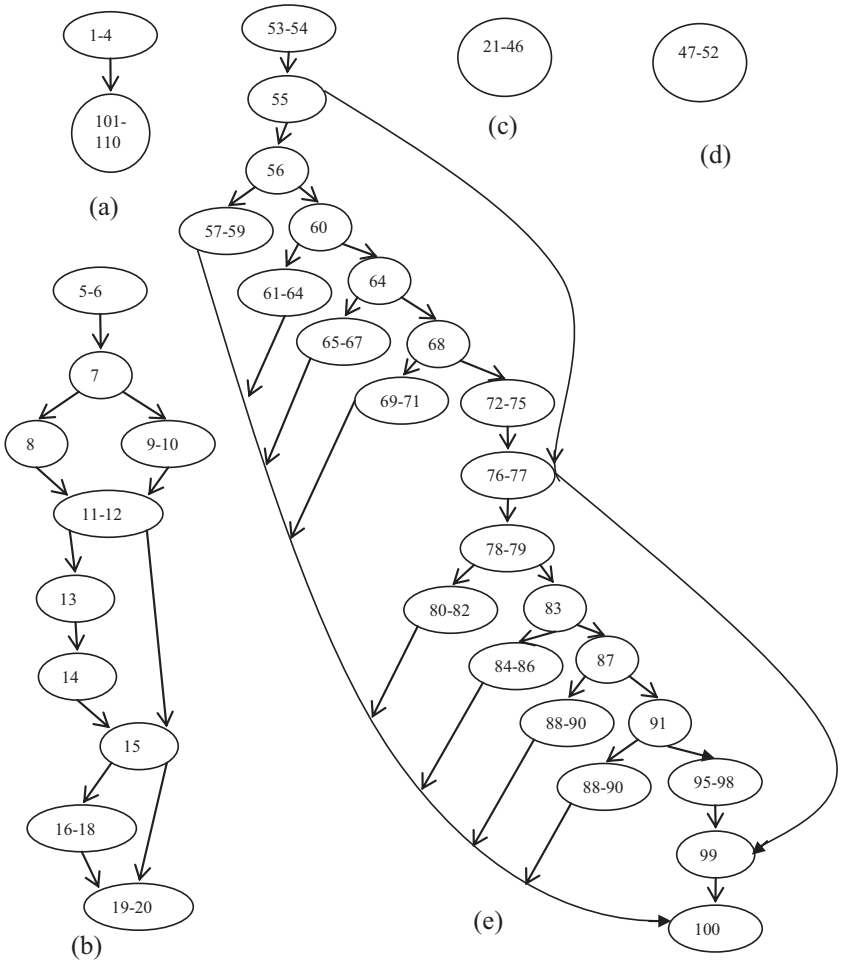

Fig. 5. DD path graph created from the function (a) float calc_vc(int age, int exp_in_yrs, int extra_working_hrs, char dept[]), (b) void insert(), (c) void display(), (d) float calctax (float totalincome) and (e) int main().

that may cause type 1 and type 2 interaction failures is shown in Tables 10 and 11 respectively. Finally, the set of interactions identified is reduced to a minimal set, which is shown in Table 12.

The output of our approach is Table 6 for case study 1 and Table 12 for case study 2. These tables show the minimal set of interactions among variables that must be tested.

5. Results and discussion

In this section, we present the result of proposed approach on the case studies taken and measure the effectiveness of our approach. To develop confidence in our approach, we compared our approach with the t-way testing. We have compared our approach to t-way testing on the following points:

1. Comparison in terms of number of interactions to be tested.
2. Comparison in terms of test set size.
3. Comparison in terms of statement and branch coverage achieved.

We further evaluated the effectiveness of our approach using mutation testing.

5.1. Comparison in terms of number of interactions to be tested

The measure of the comparison is the count of interactions required to be tested. In the case study 1, as the highest interaction strength identified for an interaction is 3; it is required to perform 3-way testing to identify interaction faults. 3-way testing would require testing of 35 triplet interactions. However, with the help of our approach, only a triplet interaction and two pairwise interactions need to be tested. Similarly, in the case study 2, as the highest interaction strength identified for an interaction is 4; it is required to perform 4-way testing to identify interaction faults. 4-way testing would require testing of 3876 quad tuple interactions. However, with the help of our approach, only a quad tuple interaction, two triples and three pairwise interactions need to be tested. Hence, with the help of proposed approach; we can reduce the count of interactions to be tested.

5.2. Comparison in terms of test set size

We performed comparison on the basis of size of the test set generated using our approach and the test set generated using basic t-way testing. As the variables can have a range of values, it is required to partition the range into a finite number of intervals. Using the equivalence partitioning technique [2], the range of variables can be divided into the following intervals as given in Table 13 (for case study 1) and Table 14 (for case study 2). For example, variable 'date' can have one of the values from 1–28, 29, 30, 31 or (−1, 32).

For case study 1, performing 2-way testing would require testing of 291 pairs and 49 test cases are required to cover these pairs. On the other hand, performing 3-way testing requires testing of

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

Table 9
c-use, p-use and r-use of variables at each node of the DD flow graph of case study 2.

Node	c-use	p-use	r-use
7		Dept	
8	Bsal		
10	Bsal		
11–12	Vc, extra_working_hours, bsal	Age	
13		Exp_in_yrs	
14	Vc, dsal		
15		Design	
17	Vc		
43–45	{{Bsal, dsal,hsal,vc},{ Totalsalary,pf,insurance }}		rv_Calc_vc()
55		Gender	
56		Totalincome	
60		Totalincome	
62	Totalincome		
64		Totalincome	
66	Totalincome		
68		Totalincome	
70	Totalincome		
74	Totalincome		
77		Gender	
79		Totalincome	
83		Totalincome	
85	Totalincome		
87		Totalincome	
89	Totalincome		
91		Totalincome	
93	Totalincome		
97	Totalincome		
104–106			{{rv_insert()},{rv_display()},{rv_calc_tax()}}

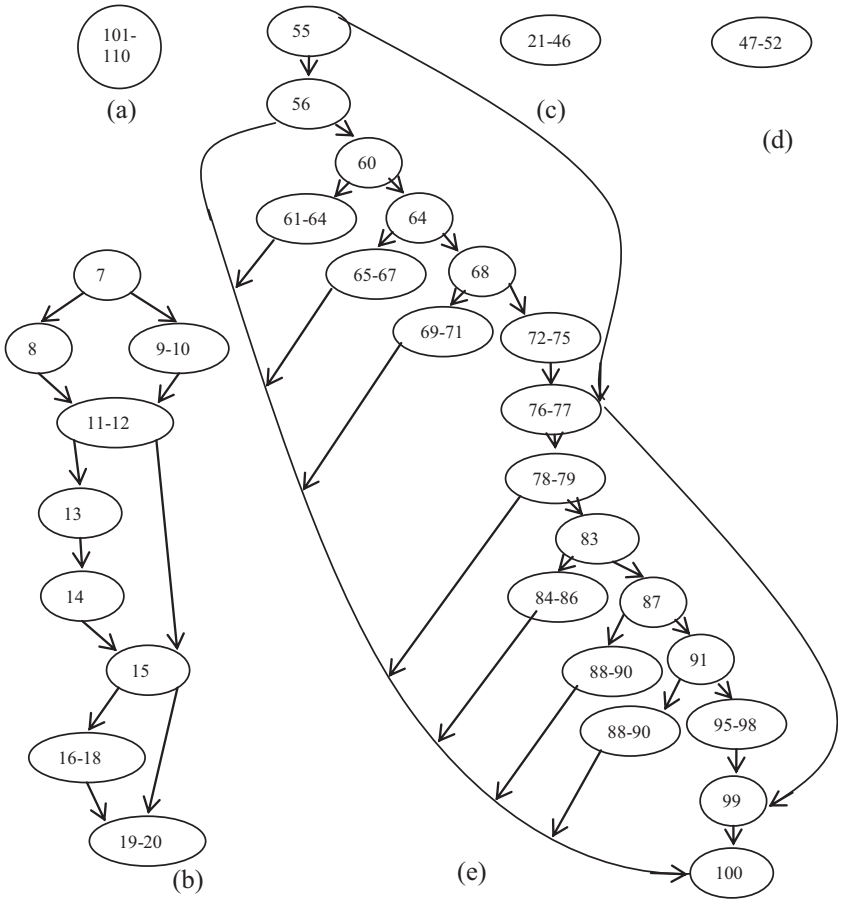

Fig. 6. RDD path graph created from the DD path graph of the function (a) float calc_vc(int age, int exp_in_yrs, int extra_working_hrs, char dept[]), (b) void insert(), (c) void display(), (d) float calctax(float totalincome) and (e) int main().

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

Table 10
Interactions among variables that may cause type 1 interaction failures of case study 2.

Path	Interactions
101–110	–
21–46	–
47–52	–
7, 8, 11–12, 13, 14, 15, 19–20	{{Dept},{age,exp_in_yrs},{design}}
7, 8, 9–10, 11–12, 13, 14, 15, 19–20	{{Dept},{age,exp_in_yrs},{design}}
7, 8, 9–10, 11–12, 13, 14, 15, 16–18, 19–20	{{Dept},{age,exp_in_yrs},{design}}
55, 56, 60, 64, 68, 72–75, 76–77, 78–79, 83, 87, 91, 95–98, 99, 100	Gender, totalincome
55, 56, 60, 64, 68, 72–75, 76–77, 78–79, 83, 87, 91, 92–94, 100	Gender, totalincome
55, 56, 60, 64, 68, 72–75, 76–77, 78–79, 83, 87, 88–90, 100	Gender, totalincome
55, 56, 60, 64, 68, 72–75, 76–77, 78–79, 83, 84–86, 100	Gender, totalincome
55, 56, 60, 64, 68, 72–75, 76–77, 78–79, 80–82, 100	Gender, totalincome
55, 56, 60, 64, 68, 72–75, 76–77, 99, 100	Gender, totalincome
55, 76–77, 99, 100	Gender
55, 56, 100	Gender, totalincome
55, 56, 60, 61–63, 100	Gender, totalincome
55, 56, 60, 64, 65–67, 100	Gender, totalincome
55, 56, 60, 64, 68, 69–71, 100	Gender, totalincome

Table 11
Interactions among variables that may cause type 2 interaction failures of case study 2.

Interactions
Vc, extra_working_hours, bsal
Vc, dsal
Bsal, dsal, hsal, vc
Totalsalary, pf, insurance

Table 12
Minimal set of interactions among variables of case study 2.

Interactions
Vc, extra_working_hours, bsal
Vc, dsal
Bsal, dsal, hsal, vc
Totalsalary, pf, insurance
Age, exp_in_yrs
Gender, totalincome

1593 triplets and 245 test cases are required to cover these triplets. Using our approach, test cases are generated such that all the identified interactions are tested. Hence, for the case study 1, using our proposed approach 14 pairs and 70 triplets are required to be tested, which requires 70 test cases. Similarly, for case study 2, performing 3-way testing would require testing of 14,306 triplets and 52 test cases are required to cover these triplets. On the other hand, performing 4-way testing requires testing of 137,803 quad tuples and 124 test cases are required to cover these quad tuples. Using our approach, test cases are generated such that all the identified interactions are tested. Hence, for the case study 2, using our proposed approach 19 pairs, 45 triplets and 81 quad tuples are required to be tested, which requires 81 test cases.

5.3. Comparison in terms of statement and branch coverage achieved

We have evaluated the effectiveness of the generated test cases using coverage metrics. Higher the coverage achieved, better is the test set. We measured statement coverage and branch coverage. Statement coverage is the ratio of number of statements executed at least once to the total number of statements. Branch coverage is the ratio of number of branches executed at least once to the total number of branches. We compiled the program and executed the test cases using GCC compiler and measured the coverage using gcov tool. As we generated test set using Genetic Algorithms [29], which are based on random numbers, each experiment was executed 10 times and the results obtained were averaged. Our test set achieved 100% statement coverage and 100% branch coverage for both the case studies.

For comparison purpose, we generated test set by considering only input parameters, namely date, month and year for case study 1 and employee's name, ID, Gender, Bsal, Dsal, Hsal, age, exp_in_yrs, extra_working_hours, dept, design, pf and insurance amount for case study 2. Test cases were generated such that all combinations of input values were tested. Test set was generated

Table 13
Variables and their possible values for case study 1.

Variable	Value 1	Value 2	Value 3	Value 4	Value 5	Value 6	Value 7
Date	1–28	29	30	31	−1, 32		
Month	Has 30 days	Has 31 days	Is February				
Year	Is leap year	Is not a leap year					
Leap	0	1					
Fmonth	0	1	2	3	4	5	6
Dow	0	1	2	3	4	5	6
Ret_fm	0	1	2	3	4	5	6

Table 14
Variables and their possible values for case study 2.

Variable	Value 0	Value 1	Value 2
Employee's name	String		
Employee's ID	Numeric value		
Designation	Manager	Team leader	Engineer
Gender	Male	Female	
Department	Accounts	Admin	Development
Basic salary	21,600	29,700	37,100
DA	<10,000	10,000–15,000	>15,000
HRA	<8000	8000–12,000	>12,000
Age	21–50 yrs	>50 yrs	
Exp_in_yrs	<20 yrs	⩾20 yrs	
Extra working hours	0	1–30	>30
PF	2000	5000	9000
Insurance	1500	2500	
Total salary	<55,000	55,000–65,000	>65,000
Total income	<65,000	65,000–75,000	>75,000
Tax	0	1–30,000	>30,000
Var. Component (vc)	<2000	2000–6000	>6000
rv_calc_vc	<2000	2000–6000	>6000
rv_calctax	0	1–30,000	>30,000

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

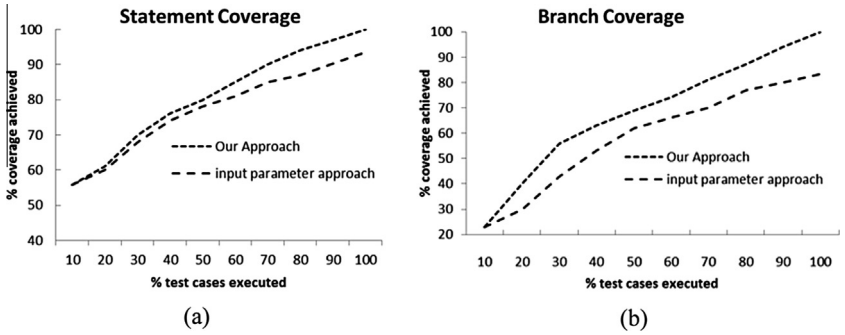

Fig. 7. Graph between percentage of test cases executed and (a) statement coverage achieved and (b) branch coverage achieved for case study 1.

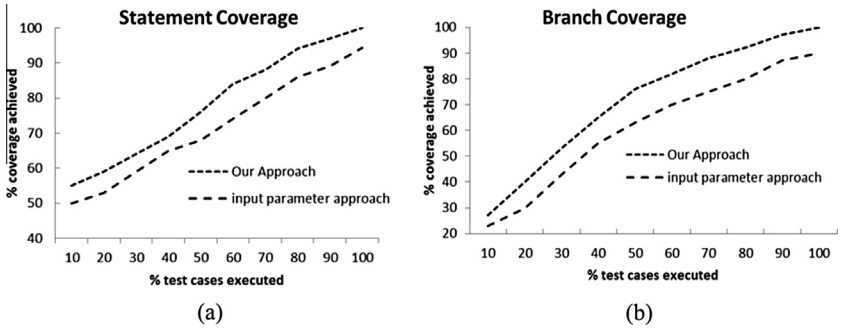

Fig. 8. Graph between percentage of test cases executed and (a) statement coverage achieved and (b) branch coverage achieved for case study 2.

using Genetic Algorithms [12]. The generated test set achieved 93.44% statement coverage and 83.33% branch coverage for case study 1. Similarly, for case study 2, the generated test set achieved 94.36% statement coverage and 90.00% branch coverage. Hence, we can conclude that our test set achieves higher coverage. Figs. 7 and 8 shows the graph between percentage of test cases executed and (a) statement coverage achieved and (b) branch coverage achieved for the case studies taken. By input parameter approach we mean basic t-way testing approach, i.e. it considers all the input parameters of a system and for a value of *t*, as given by the tester, generates a test set such that all the t-way interactions are getting covered. Test set generated using our approach is able to achieve higher coverage at a faster rate as compared to the test set generated using input parameters. It is to be noted that success of equivalence partitioning method depends on the selection of partitions created. A careful examination of the program would help in creating better partitions. Better the partitions defined, better is the coverage achieved. However, as we are using the same partitions for both the approaches, results for the two approaches would be affected in same manner.

We further performed mutation testing. Mutants were introduced manually. To avoid a potential source of bias, all possible mutations were generated either by making changes in the arithmetic or logical operators. Then 30 mutations were randomly selected. We seeded 30 mutations into our program for case study 1 and 2. Each mutation was injected in the respective program independently and the test set was executed. The generated test set was able to kill all the mutants for case studies

From the case studies, we can conclude that rather than considering only input parameters, our approach identifies parameters from the source code that are significantly involved in the interactions. Hence, we can state that our approach attempts to model the input space and identifies the possible interactions. As can be concluded from the results, test set generated using input parameters does not achieve significant coverage. In contrast, test set generated using our approach is able to achieve (near) 100% coverage. Hence, we can state that test set generated using our proposed approach covers all the identified interactions and achieves (near) 100% statement and branch coverage.

6. Threats to validity

Empirical experiments are prone to threats to validity. Attempts have been made to reduce them. In this section, we discuss the major threats. For any structured program, it was difficult to identify a program having large number of input parameters. The benchmark problems taken by [6] have 4–10 variables only. We have considered only two of the benchmark problems. We did our own implementation of the benchmark problems. The case studies taken have 7 and 19 parameters only. Secondly, we have considered mutation operators by changing arithmetic and logical operators only. Other mutation operators such as addition or deletion of a statement, replacement of a variable may be considered as a future work. Thirdly, we have manually generated and added mutants. To avoid bias, we considered all possible variants of arithmetic and logical operators and randomly selected 30 mutations from them. Lastly, we claimed that our approach is scalable for large sized programs. It is based on the fact that as we are considering a single statement at a time, increasing the number of parameters would not affect the approach, making our approach scalable for programs involving large number of parameters. We also expect that our approach does not suffer from path explosion problem. Our claim is based on the fact that, rather than generating all the possible paths in the program, we have generated independent paths. Also, instead of creating a DD path graph for a complete program, we have utilized the modular property of a structured program. We have created a graph for each function, leading to finite number of (depending on number of functions defined in the program) small sized graphs.

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008

7. Conclusion and future work

In this paper, we have proposed a novel approach using data flow techniques that reduces the count of interactions to be tested. The probable interactions that may cause the failure to occur are identified at an earlier stage, i.e. even before the occurrence of failures. Our work tries to identify interaction faults. With the help of data flow technique, from the DD path graph, interactions of variables are identified. As a result, rather than testing all the possible t-way interactions, only the identified interactions are tested. The approach is able to handle structured programs where the program is divided into modules known as functions. Various interdependencies between functions, such as a function being called by main function, a function being called by another function, a function returning a value and a function returning void have been considered. The effectiveness of the proposed approach is shown by taking two case studies. Experimental results indicate that the proposed approach in both the case studies achieves a significant reduction in the count of interactions to be tested. Further, test set generated using our approach is able to achieve 100% statement and branch coverage. Also, the test set is able to kill all the mutants that have been manually introduced in the program. Although the initial results are presented for a medium sized program, the approach can be extended to large sized systems and object oriented systems. As a part of future work, we will automate our approach and develop a tool to support our approach. Further, we will study time complexity of our approach using the tool and handle aliases of variables.

Appendix A

In this appendix, we provide the source code of the problems taken as case study. Case study 1 is a C Program that accepts Year, Month and Date from the user and displays the day of the month [28].

```
1.      int isdatevalid(int month, int date, int year) {
2-32    //to check if the input is a valid date
33.     }
35.     int fm(int date, int month,int year){
36.     int fmonth,leap;
37-47   /*code to check if it is a leap year. Returns 1 for
        leap year & 0 for non-leap year. Calculates f(m)
        value and converts fmonth value to value 0 to 6
        using remainder function. Returns fmonth */
48.     }
50.     int day_of_week(int date, int month, int year){
51.     int dow; //day of week
53.     int ret_fm= fm(date,month,year);
54.     dow = 1.25 * (year%100) + ret_fm+date-
        2*((year/100)%4); //method of calculating
        weekday for Gregorian
56.     dow = dow % 7; // convert to value 0 to 6
57.     switch (dow){
58-72   //code to print day of week using switch
73.     }
74.     return 0;
75.     }
77.     void main(){
78.     int date,month,year;
79-85   //read input from user
86.     if (!isdatevalid(date, month, year))
87.       return 0;
88.     else
89.       day_of_week(date,month,year);
90.     getch();
91.     }
```

Case study 2 is a C Program that accepts various employee details and calculates income tax.

```
1.      int emp_id,extra_working_hours;
2.      char emp_name[10], add[10], gender, dept[10],
        design[10];
3.      float bsal, dsal, hsal, ma, pf, insurance, vc,
        totalsalary, totalincome;
4.      int age, exp_in_yrs;
5.      float calc_vc(int age, int exp_in_yrs, int
        extra_working_hrs, char dept[])
6.      {
7.      if(dept=="admin")
8.          vc=bsal*0.1;
9.      else
10.         vc=bsal*0.08;
11.     vc=vc+extra_working_hrs*bsal*0.01;
12.     if(age>50)
13.         if(exp_in_yrs>20)
14.           vc=vc+dsal*0.05;
15.     if(design=="manager")
16.     {
17.         vc+=5000;
18.     }
19.     return vc;
20.     }
21.     void insert()
22.     {
23-42   //code to read input parameter values
43.     vc=calc_vc(age,exp_in_yrs,extra_working_
        hours,dept);
44.     totalsalary = (bsal + dsal + hsal + vc)/2;
45.     totalincome = totalsalary*12 + (pf+insurance)/2;
46.     }
47.     void display()
48.     {
49-51   //code to print Employee Name, Total Salary
        and Total Income
52.     }
53.     float calctax(float totalincome)// to return the
        calculated tax
54.     {
55.     if (gender=='m')
56-76.  ./* if total income is less than 150000, then no tax
        is to be paid
        else if total income ranges between 150000 and
        300000, then tax is 10% of total income
        else if total income ranges between 300000 and
        500000, then tax is 20% of total income
        else if total income ranges between 500000 and
        1000000, then tax is 30% of total income
        else tax is 40% of total income */
77.     if(gender == 'f')
78-98.  /* if total income is less than 180000, then
        no tax is to be paid
        else if total income ranges between 180000
        and 300000, then tax is 10% of total income
        else if total income ranges between 300000 and
```

```
        500000, then tax is 20% of total income
        else if total income ranges between 500000 and
        1000000, then tax is 30% of total income
        else tax is 40% of total income */
99.     }     return 0;
100.    }
101.    int main()
102.    {
103.    float tax;
104.    insert();
105.    display();
106.    tax = calctax(totalincome);
107.    //code to print calculated Tax
108.    getch();
109.    return 0;
110.    }
```

References

[1] R.S. Pressman, Software Engineering: A Practitioner's Approach, third ed., McGraw Hill, New York, 1992.
[2] P.Aditya Mathur, Foundations of Software Testing, 2/e, Pearson Education, India, 2008.
[3] G.J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, third ed., Wiley, New York, NY, USA, 2011.
[4] D.R. Chicago Kuhn, R.N. Kacker, Y. Lei, Practical Combinatorial Testing, NIST Special Publication, 2010. 800-142.
[5] D.M. Cohen, S.R. Dalal, J. Parelius, G.C. Patton, The combinatorial design approach to automatic test generation, IEEE Softw. 13 (5) (1996) 83–88.
[6] S. Varshney, M. Mehrotra, Search-based test data generator for data-flow dependencies using dominance concepts, branch distance and elitism, Arabian J. Sci. Eng. (2015) 1–29.
[7] S. Sabharwal, M. Aggarwal, Identifying interactions for combinatorial testing using data flow techniques, in: SIGSOFT, vol. 39, issue 6, ACM, New York, NY, 2014.
[8] C. Nie, H. Leung, A survey of combinatorial testing, ACM Comput. Surv. J. 43 (2) (2011) 11:1–11:29.
[9] L.G. Hernandez, N.G. Valdez, J.T. Jimenez, Construction of mixed covering arrays of strengths 2 through 6 using a tabu search approach, Discrete Math.: Algorithms Appl. 4 (3) (2012) 1–20.
[10] M.B. Cohen, C.J. Colbourn, A.C.H. Ling, Augmenting simulated annealing to build interaction test suites, in: Proc. of the 14th International Symposium on Software Reliability Engineering (ISSRE'03), IEEE Computer Society, Los Alamitos, CA, 2003, pp. 394–405.
[11] B.S. Ahmed, M.A. Sahib, M.Y. Potrus, Generating combinatorial test cases using simplified swarm optimization (SSO) algorithm for automated GUI functional testing, Eng. Sci. Technol. Int. J. 17 (4) (2014) 218–226.
[12] P. Flores, Y. Cheon, PWiseGen: generating test cases for pairwise testing using genetic algorithms, in: IEEE International Conference on Computer Science and Automation Engineering (CSAE), IEEE, Shanghai, 2011, pp. 747–752.
[13] Y. Wang, H. Wu, Z. Sheng, A prioritized test generation method for pair-wise testing, TELKOMNIKA Indonesian J. Electr. Eng. 11 (1) (2013) 136–143.
[14] T. Berling, P. Runeson, Efficient evaluation of multifactor dependent system performance using fractional factorial design, IEEE Trans. Softw. Eng. 29 (9) (2003) 769–781.
[15] R. Krishnan, S.M. Krishna, P.S. Nandhan, Combinatorial testing: learnings from our experience, in: SIGSOFT Software Engineering Notes, vol. 32, issue 3, ACM, New York, NY, USA, 2007, pp. 1–8.
[16] S. Vilkomir, B. Amstutz, Using combinatorial approaches for testing mobile applications, in: IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, Cleveland, OH, 2014, pp. 78–83.
[17] A.W. Williams, R.L. Probert, A practical strategy for testing pair-wise coverage of network interfaces, in: Proc. of the 7th International Symposium on Software Reliability Engineering (ISSRE'96), IEEE Computer Society, Los Alamtos, CA, 1996, pp. 246–254.
[18] M.B. Cohen, M.B. Dwyer, J. Shi, Interaction testing of highly-configurable systems in the presence of constraints, in: Proc. of the International Symposium on Software Testing and Analysis (ISSTA'07), ACM, New York, 2007, pp. 129–139.
[19] C. Yilmaz, M.B. Cohen, A.A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, IEEE Trans. Softw. Eng. 32 (1) (2006) 20–34.
[20] C. Yilmaz, M.B. Cohen, A.A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, IEEE Trans. Softw. Eng. 32 (1) (2006) 20–34.
[21] X. Niu, C. Nie, Y. Lei, A.T.S. Chan, Identifying failure-inducing combinations using tuple relationship, Proc. of IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Luxembourg, 2013, pp. 271–280.
[22] M. Grindal, J. Offutt, Input parameter modeling for combination strategies, in: Proc. of the 25th conference on IASTED International Multi-Conference, ACTA Press, Anaheim, CA, USA, 2007, pp. 255–260.
[23] P. Satish, K. Sheeba, K. Rangarajan, Deriving combinatorial test design model from UML activity diagram, in: Sixth International Conference on Software Testing, Verification and Validation Workshops, IEEE, Luxembourg, 2013, pp. 331–337.
[24] P. Satish, A. Paul, K. Rangarajan, Extracting the combinatorial test parameters and values from UML sequence diagrams, in: IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, Cleveland, OH, 2013, pp. 88–97.
[25] C. Cheng, A. Dumitrescu, P. Schroeder, Generating small combinatorial test suites to cover input output relationships, in: Proc. of the 3rd International Conference on Quality Software (QSIC'03), IEEE Computer Society, Los Alamitos, CA, 2003, pp. 76–82.
[26] P.J. Schroeder, B. Korel, Black-box test reduction using input-output analysis, in: Proc. of the 2000 ACM SIGSOFT international symposium on Software Testing and Analysis, ACM SIGSOFT Software Engineering Notes, vol. 25, issue 5, ACM, New York, 2000, pp. 173–177.
[27] P.C. Jorgensen, Software Testing: A Craftsman's Approach, CRC Press, 2013.
[28] A. Thakur, Calender Program in C Programming Language, 2013. From <http://amitthakur744.blogspot.in/2013_08_01_archive.html>.
[29] B.S. Ahmed, Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing, Eng. Sci. Technol. Int. J. (2015), http://dx.doi.org/10.1016/j.jestch.2015.11.006.

Please cite this article in press as: S. Sabharwal, M. Aggarwal, A novel approach for deriving interactions for combinatorial testing, Eng. Sci. Tech., Int. J. (2016), http://dx.doi.org/10.1016/j.jestch.2016.05.008